Discussion 10 (11/12)

- Reminder that the quiz has been released!
- Review last week's discussion if you didn't finish and the students have questions
- Also note that you don't have to cover all of the examples!! Just a few should be good for students to understand the concepts, but if you have time at the end of class feel free to return to any examples you may have skipped
- More Advanced Lambda **(Located in Disc 10 - Lambda Calculus and Rust Basics)**
    - Recall from last week, Beta Normal Form is a reduction of the Lambda expression where it cannot be reduced further
    - To beta reduce, recall that function application is left associative, and if possible, bring a variable into a lambda
    - Example 1
        - This expression cannot be reduced any more, because we would be applying the λx expression to x, instead of the λy to λx, due to left associativity
        - Thus, it is already in normal form
    - Example 2
        - In this expression, again due to left associativity, we can apply the λy function to λx
        - Thus, in λx, any place we see the x variable, we replace it with the λy expression, getting (λy. y y) (λy. y y)
        - Applying a reduction again, we see that we have reached an infinite loop, so we can write "Not Possible"
    - Example 3
        - In this expression, we apply x to λa and replace any a's in the function with x's
        - Since there are no a's, we will substitute nothing
        - The answer is b
    - Now for some encodings examples - we will always provide the encodings, so it's similar to beta reduction with words and lambda expressions
    - Basically, show that the left side beta reduces to the right hand side
    - Note that you don't always have to expand everything out
    - Example 1
        - not (not true) = not ((λx. ((x false) true) true)
        - The parentheses actually group the second not and true together, so we can apply true to λx here. Be very cautious about the parenthese in this regard
        - not (not true) = not ((λx. ((x false) true) true) = not ((true false) true) = not (((λx.λy.x) false) true)) by replacing true with its encoding
        - Again, we can apply false in our lambda function, giving us not ((λy. false) true)
        - We apply true in our lambda function to get not false
        - not false = (λx. ((x false) true)) false = ((false false) true) by applying false into our lambda

        - ((false false) true) = ((λx.λy.y) false) true) = (λy.y) true = true!
- Rust Basics **(Located in Rust_Notes)**
    - Rust is functional and imperative both
    - Rust Types
        - Just some examples of data types in Rust
        - i32 and i64 are integer examples, f32 and f64 are float examples
        - There are more, like unsigned int, booleans, etc.
        - Arrays can be declared using simple square brackets: [T]
        - Note that arrays must be homogenous like in OCaml, but they could be of any type
        - Tuples do not need to be homogenous, and can take multiple types but only have a fixed size. For instance, (2, 3.4) takes an int and a float
        - There are string types as well: String and str
        - String is similar to Java's StringBuffer, and can be mutated to any size
        - str is a string slice (like a string array of fixed size), and it is read-only and borrow-only
        - We can convert an str to a String using to_string()
        - For example, str: "hello world" and String: "hello world".to_string()
    - Borrowing and references are the whole point of Rust, since the goal is memory safety
    - It checks for unsafe memory usage during compile time, so you can't even run unsafe code
    - It does this through the concept of scopes and borrowing
    - Borrowing and References Example 1
        - In this example, we have x declared as a String, "hello"
        - x is thus the owner of "hello"
        - We can move ownership of the string to y by running y = x, which copies the data from x into y and makes y the owner
        - This means that x is no longer a valid owner of "hello", so accessing x in the print statement, for example, would fail!
    - Borrowing and References Example 2
        - Here the curly braces create an inner scope, in which x and y have been declared
        - Thus, trying to access x or y will fail outside the curly braces, since it is outside the scope
        - The first print statement within the scope will run fine, the second one outside will fail
        - Because of scopes, Rust has very efficient garbage collection - as soon as you leave a scope, delete anything that was limited to that scope
    - Borrowing and References Example 3
        - x is again a String "hello"
        - y is a reference to "hello", and has type &String. Because it is borrowing "hello", x is still the owner of it

        - Now, both dereferecing y and directly accessing x are valid, since x is the owner and y is a reference
        - y cannot be used to change “hello”, though; it would cause an error
    - Borrowing and References Example 4
        - If we declare y as a mutable reference, then we can use it to change “hello”
        - Meanwhile, z and w are immutable borrows
    - You can have any number of borrows of the same variable, or you can have one mutable borrow
    - This is important for thread safety - we can guarantee no more than one thread is writing at any given time
    - Coding Exercise 1
        - To declare variables, use syntax similar to OCaml. In this case, we will want to change the value of sum, so we must declare it as mutable
        - Range is declared just like in Ruby
        - Also like in Ruby, we don’t need a return keyword, the last line without a semi-colon will automatically be returned. For a unit (void) function, all lines should have semi-colons
        - Answer:
```
pub fn sum_evens(i, j) {
        let mut sum = 0;
        for k in i..j {
                if k % 2 == 0 {
                        sum += k;
                }
        }
        sum
}
```
    - Coding Exercise 2
        - In Rust we can specify when we want structures like arrays to be mutable
        - We could use arr.iter(), but this will not work here because it would only borrow the array immutably. Thus, we need to manually iterate using the indices
        - Note that the array is being borrowed, but the array itself is not being passed in. This is to ensure that when raise_1 returns, the original owner of arr still owns it.
        - Also note that the borrow must be mutable, since we are making changes in place
        - Answer:
```
pub fn raise_1(arr) {
        for i in 0..arr.len() {
                arr[i] += 1;
        }
}
```